PATENT ABSTRACTS OF JAPAN

(11)Publication number : 07-306790
(43)Date of publication of application : 21.11.1995

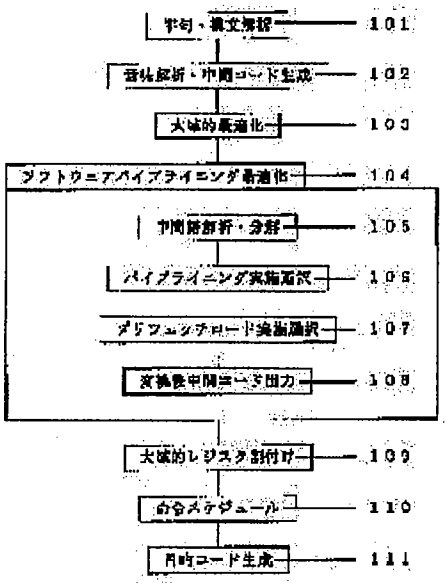
(51)Int.Cl. G06F 9/45
G06F 9/38
G06F 12/08

(21)Application number : 06-100466
(22)Date of filing : 16.05.1994

(71)Applicant : HITACHI LTD
(72)Inventor : KAWASAKI TORU

(54) LOOP OPTIMIZATION SYSTEM

(57)Abstract:
PURPOSE: To facilitate application by making a compiler automatically judge whether optimization by the issue of a prefetch load instruction effective to the dynamic reduction of a cache miss penalty and perform the optimization.
CONSTITUTION: Through an analytic path (105) for software pipelining optimization, information the number and data type of objects and the number of necessary registers which are required to apply prefetch loading is gathered, After a loop to be pipelined is selected (106), a developing method is determined by prefetch loading execution selection in 107 so that large-area register allocation (109) is not impeded; and an expansion number is divided by the number of objects so as to equalize the distance between preloading and preloading, thereby dividing parts as many as the objects. Then an intermediate word for prefetch loading indication is interposed right before each of divided part at the time of intermediate code output (108) after pipelining conversion.



Japanese Unexamined Patent Application Publication No. 7-306790

SPECIFICATION <EXCERPT>

[0015]

[Embodiment] Hereinafter, the embodiment of the present invention shall be described with reference to the Drawings. FIG. 1 shows an embodiment of the present invention. As shown in FIG. 1, a compiler advances through lexical analysis and syntax analysis of a source code (101), semantic analysis and intermediate code generation (102), and global optimization (103). The processes up to this point and processes from the global register allocation (109) onward are the same as in a conventional method. Furthermore, in this example, software pipelining optimization (104) is performed after the global optimization.

[0016] The present invention adds improvements to the process of software pipelining optimization (104), and describes the case of inserting prefetch loads, using a code fragment in FIG. 2. Optimization of such prefetch load issuing is performed only when there is an option specification by a user, and is not applied when the number of loops to be executed is small. It is to be noted that, in this example, it is assumed that the float-type is 4-byte, the cache line length is 32 bytes, and there are 32 each of the general-purpose registers and floating-point number registers. The loop iteration number N is assumed to be unknown at the time of compiling.

[0017] In the software pipelining optimization process (104), first, intermediate language analysis (105) is performed, and whether or not to perform optimization is determined in the pipelining implementation selection in 106, based on a control variable or the loop structure. The code fragment in FIG. 2 becomes an intermediate code language such as that shown in FIG. 3 by the stage in which code scanning is performed on the intermediate code language in 105. When a sequence having a control variable as an additional character or a pointer variable involving referring to the contents to be updated in the loop are found during the scanning, this is counted as a prefetch load subject. In the example in FIG. 2, sequences a and c correspond to this, and thus there are two subjects. At the same time, in this path,

the required number of registers is calculated based on the number or variables and constants. In the example in FIG. 2, the floating-point number registers are under stricter conditions, registers are needed for the targets (301, 302 in FIG. 3) for multiplication with two loads and for a variable b. Since the variable b can be considered as a constant within the loop, the same register can be allocated regardless of the number of expansions, and thus the variable b is counted as a constant.

[0018] Based on the information collected in 105, whether or not to perform pipelining optimization is determined in the pipelining implementation selection (106) process. When the loop is to be optimized as a result, the insertion method for the prefetch load is determined in 107, based on the information collected up to this point. The operational flow for 107 is shown in FIG. 4. The detailed processes are performed in the order shown in [1] to [3] below, and after 107 ends, a converted intermediate code inserted with an intermediate code specifying the prefetch load is outputted in the converted intermediate code outputting (108) process.

[0019] [1] The expansion method is determined by comparing the optimal number of expansions and the limit value for the number of expansions coming from the register resource. First, the optimal number of expansions is calculated in 401 by dividing the line length of the cache by the width of the prefetch load subject region referred to per iteration. In the example in FIG. 2, the line length is 32 bytes and the 4-byte sequence members are accessed one at a time, and thus 34/4 is 8. Next, the expansion limit value is calculated in 402 based on the required number of registers in 402 so that there is no shortage of registers in the subsequent global register allocation (109). When the number of variables is v, the number of constants is c, and the number of registers that can be allocated is m, the maximum number of expansions is none other than the largest natural number n which satisfies the inequality expression below.

$$n \times v + c \leq m$$

In the example in FIG. 2, since v=3, c=1, and m=32, the maximum number of expansions n becomes 10. Since the comparison of this result 403 is satisfied, loop expansion is performed for the optimal number of expansions (404). Even when assuming the case where the

comparison is not satisfied, the reason why the prefetch load requires loop expansion is different from the conventional case, and thus, by repeating the possible number of expansions as shown in FIG. 5, the code for the optimal number of expansions is repeated in order to enable execution in a single iteration (405). FIG. 5 is an example in which the optimal number of expansions 8 is implemented by repeating the codes for four expansions (501, 502) twice, and use of the same register set in 501 and 502 is made possible.

[0020] [2] The inserting position is determined from the number of prefetch loads. First, the number of prefetch loads is checked (406), and the prefetch load is placed at the beginning of the loop when there is only one (407). When there are plural prefetch loads, the number of expansions is divided by the number of prefetch loads (408), and the prefetch loads are positioned in equal intervals as much as possible. Since there are two prefetch loads in the example in FIG. 2, codes are positioned at four expansions each, and prefetch loads (601, 603) corresponding to sequences a and c are respectively inserted at the beginning of each part (602, 603), as shown in FIG. 6. With this, the inserted prefetch loads will be in roughly equal intervals, and by cutting the units of the command schedule there, such interval can be maintained even without the command scheduler (FIG. 1, 110) having to pay any attention from here on.

[0021] However, this method is one example for implementation, and it is also possible to have a design in which a command schedule (110) re-positions prefetch loads within the loop in equal intervals.

[0022] [3] Next, the width of the prefetch is determined in 409. Although this is highly dependent on the hardware mechanism, normally, it is sufficient to have leeway just for cache miss processing to finish by the time a prefetched location is accessed. However, when there is hardware support which invalidates the command in the case where a prefetch load appears during another cache miss processing, a greater width of about one line length is taken.

[0023] It is to be noted that, in this example, it is assumed that the command parallelism supported by the subject calculator is comparatively low. As such, since the number of expansions requiring software pipelining is smaller than the subject of the comparison in 403,

this is not taken into consideration here. Furthermore, for the same reason, in the global register allocation (109), the required number of registers is estimated under the assumption that the smallest number of registers is to be allocated. For a calculator having a higher command parallelism, it is necessary to take into consideration that these balances are significantly changed.

[0024]

[Advantageous Effects of Invention] According to the present invention, since optimization through the issuance of a prefetch load can be automatically performed by a compiler, it is possible to automatically reduce processor stalling which occur due to a cache miss, and significantly improve the execution performance of an object code.

[Brief Description of Drawings]

FIG. 1 is a structure diagram of a compiler utilized in the present invention.

FIG. 2 is a diagram showing a source program for describing a specific example of the present invention.

FIG. 3 is a diagram showing a model of intermediate code for the source program in FIG. 2.

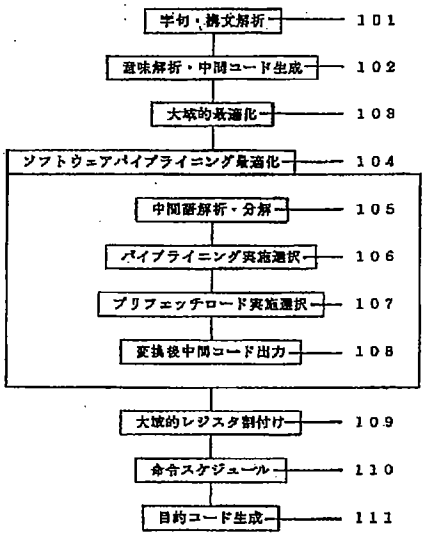
FIG. 4 is a structure diagram of a part which determines prefetch load implementation.

FIG. 5 is a diagram showing an expansion method for the case where the number of loop expansions required for optimal load prefetch outputting cannot be secured due to a restriction on the register resources.

FIG. 6 is a diagram showing a model of converted intermediate code for the source program in FIG. 2.

【図1】 FIG. 1

図 1



101 Lexical and syntax analysis
102 Semantic analysis and intermediate code generation
103 Global optimization
104 Software pipelining optimization
105 Intermediate language analysis and break down
106 Pipelining implementation selection
107 Prefetch load implementation selection
108 Converted intermediate code outputting
109 Global register allocation
110 Command schedule
111 Target code generation

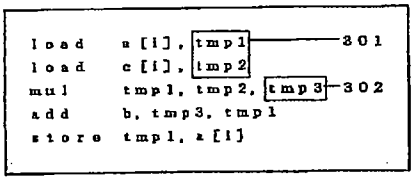
【図2】 FIG. 2

図 2

```
float    a[N], c[N], b
int      i;
                :
                :
for (i = 0; i < N; i++) {
    a[i] += b * c[i];
}
                :
```

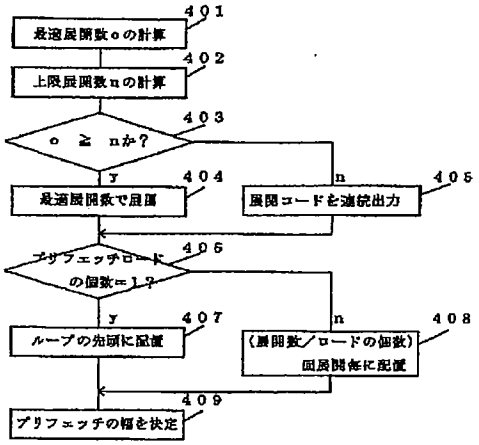
【図3】 FIG. 3

図 3



【図4】 FIG. 4

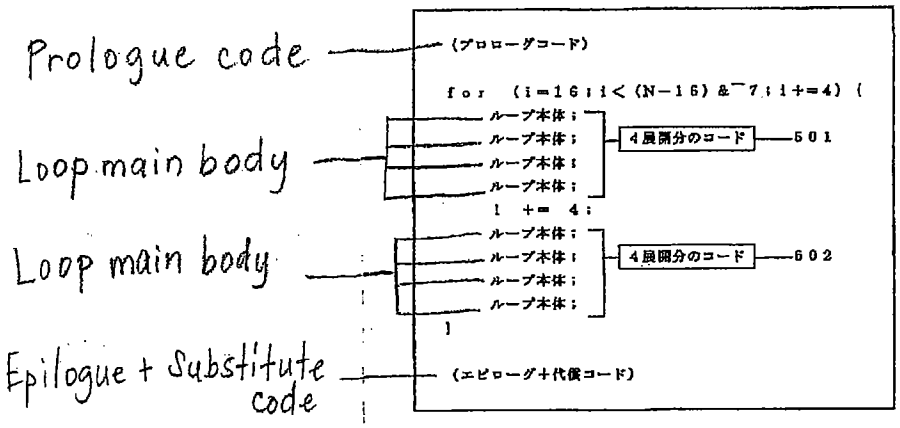
図 4



401 Calculate optimal number of expansions o
402 Calculate upper limit on number of expansions n
403 o ≥ n?
404 Expand according to optimal number of expansions
405 Consecutively output expansion codes
406 Number of prefetch loads = 1?
407 Position at beginning of loop
408 (Number of expansions/number of loads)
Position per iteration of expansion
409 Determine width of prefetch

【図5】
図 5

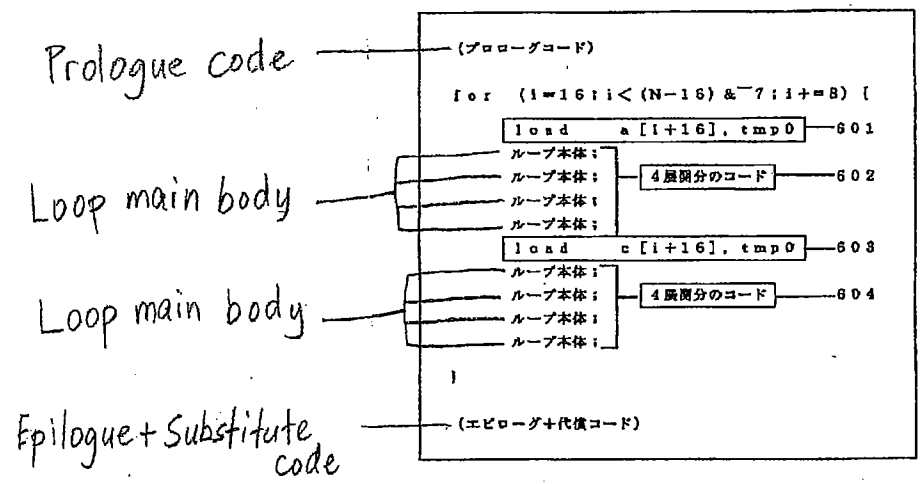
FIG. 5



501, 502 Code for 4 expansion

【図6】
図 6

FIG. 6



602, 603 Code for 4 expansion

(19)日本国特許庁（ＪＰ）　(12)**公開特許公報（Ａ）**　(11)特許出願公開番号

特開平7－306790

(43)公開日　平成７年(1995)11月21日

(51)Int.Cl.[6]	識別記号	庁内整理番号	ＦＩ	技術表示箇所
Ｇ０６Ｆ　9/45				
9/38	３３０　Ｅ			
12/08		Ｄ　7608－5Ｂ		
		7737－5Ｂ	Ｇ０６Ｆ　9/44　３２２　Ｇ	

審査請求　未請求　請求項の数１　ＯＬ　（全　５　頁）

(21)出願番号　特願平6－100466

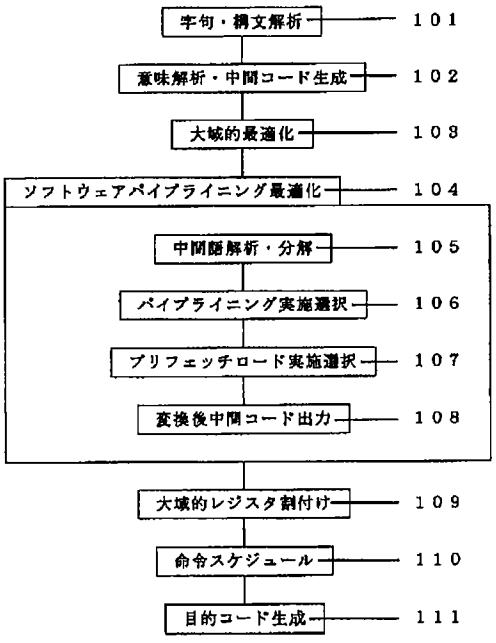
(22)出願日　平成６年(1994)５月16日

(71)出願人　000005108
株式会社日立製作所
東京都千代田区神田駿河台四丁目６番地
(72)発明者　川崎　徹
神奈川県横浜市戸塚区戸塚町5030番地　株式会社日立製作所ソフトウェア開発本部内
(74)代理人　弁理士　小川　勝男

(54)【発明の名称】　ループ最適化方式

(57)【要約】
【目的】キャッシュミスペナルティーを動的に削減するのに有効なプリフェッチロード命令の発行による最適化を、コンパイラが自動的に適否を判断して実施できるようにすることによって、適用を容易にする。
【構成】ソフトウェアパイプライニング最適化の解析パス(105)で、プリフェッチロードの適用に必要な対象の数やデータ型、及び必要レジスタ数といった情報を収集する。パイプライニング対象ループを選択(106)した後、107のプリフェッチロード実施選択で、大域的レジスタ割付け(109)を阻害しないように展開方法を決定し、プリフェッチロード同士の距離を概ね等間隔にする為に、展開数を対象数で割って、対象数個の部分に分ける。その後、パイプライニング変換後の中間語コード出力(108)時に、分割された各部分の直前にプリフェッチロード指示の中間語を挿入する。

図　1



1

【特許請求の範囲】

【請求項１】パイプライン制御を行なう計算機を対象とするコンパイラにおいて、コンパイラがループにソフトウェアパイプライニングを施す時点で、データプリフェッチのロード命令をオブジェクト中に挿入することにより、実行時のキャッシュミスペナルティーを削減する方法。

【発明の詳細な説明】

【０００１】

【産業上の利用分野】情報処理に係わり、特に、アプリケーションプログラムの実行性能を向上させる方式に関する。

【０００２】

【従来の技術】これまでに、パイプライン制御を行う計算機に対するプログラムの実行性能を向上するものとして、ソフトウェアパイプライニング最適化が適用されてきた。ソフトウェアパイプライニングとは、与えられた独立な機能ユニット間の並列性を利用するために命令列を再構成する事であり、このことで命令実行をオーバーラップさせ、性能向上を図るものである。この技術を実際のコンパイラに適用した例として、Sridhar Ramakrishman,"Software Pipelining in PA-RISC Compilers",Hewlett-Packard Journal,June 1992等がある。

【０００３】

例えば次のコード片

```
for (i=0;i<N;i++){
    a[i]+=b*c[i];
}
```

に対する繰返し中のオブジェクト

```
load  a[i],R1
load  c[i],R2
mult  b,R2,R3
add   R1,R3,R4
store R4,a[i]
```

では、レジスタが依存するため、たとえ機能ユニットが十分にあってもこれらの命令を同時に実行することはできない。そこで、このオブジェクトをロード、掛け算、足し算、ストア、の4つのグループに分けた上で、4回展開したループの中で並べ替えると、繰り返し1回に相当するオブジェクトは次のようになり、レジスタの依存をなくすことができる。

【０００４】

```
load  a[i+3],R1
load  c[i+3],R2
mult  b,R3,R4
add   R5,R6,R7
store R8,a [i]
```

この最適化によって、必要レジスタ数とオブジェクト量が増加するが、機能ユニットを効率的に利用することが可能になる。

2

【０００５】また従来、キャッシュに載らないデータをアクセスする場合の実行性能向上を図る手法として、プリフェッチロードを用いた最適化が提唱されてきた。これは後方で用いるデータのロード命令を予め発行することで、このデータをメモリからキャッシュに持ってくるまでの処理を、命令実行にオーバーラップさせて、動的なキャッシュミスペナルティーを削減する手法である。この方式を適応した事例として、Anne Rogers, Kai Li,"Software Support for Specculative Loads",ASPLOS V,Sept '92などが報告されている。この論文は、例えばレジスタm個を使う次のループ

```
for (i = 0; i < n;i++){
    ld     R1,A1;
           :
    ld     Rm,Am;
    f(i,R1,...,Rm)
}
```

を変形して、

```
sld    R01,A1;
         :
sld    R0m,Am;
for (i = 0; i < n;i++){
    sld    R11,A1;
           :
    sld    R1m,Am;
    f(i,R01,...,R0m)
    if ( ++ i)= n ) break;
    sld    R01,A1;
           :
    sld    R0m,Am;
    f(i,R11,...,R1m)
}
```

のようにすることで、ロードしたレジスタを使用する迄の時間を長くして、この間でキャッシュミス処理と命令実行をオーバーラップさせることが可能である、と報告している。

【０００６】しかし、この論文は、幾つかのテストケースについてプリフェッチロードをハンドパッチで挿入して、その効果を検証したものであり、実際のコンパイラに対する適応については言及がない。

【０００７】

【発明が解決しようとする課題】ワーキングセットが大きいアプリケーションプログラムでは、アクセスするデータがキャッシュに載らなかったときにかかる処理時間が全体の実行時間をも支配する傾向があるため、プリフェッチロードの発行によるキャッシュミス処理時間の削減は、性能向上に大きな効果がある。しかし、従来技術では、プリフェッチロードをコンパイラに発行させることには配慮されていなかったため、実際にコンパイラのユーザがこの最適化を適用させる事はほとんど不可能で

あった。

【０００８】本発明は、上記欠点を解消し、ソフトウェアパイプライニングの実施ループに対して、コンパイラが適否を自動的に判断した上でプリフェッチロードを発行することで、この最適化を効率的に適用することを目的とする。

【０００９】

【課題を解決するための手段】本方式では上記の目的を、ソフトウェアパイプライニング最適化を行う処理に手を加えることで実現する。この処理は、ソースプログラムに対応する中間語を走査しながら、ループの制御変数やサイズを計算するとともに、依存の見つかった時点で中間語を切り分けていき、走査後これらの情報に基づいて、最適化実施の可否を判断し、適当と判断したループについては切り分けた中間語コードを再構成することによって、パイプライニング変換した中間語コードを出力する。

【００１０】本方式では、中間語走査時に、順次アクセスする独立した領域を見つけると、これをプリフェッチロードの対象として、その数をカウントする。またこのとき、中間コードから推定した必要レジスタ数も計算する。ループがパイプライニング可能と判断された後に、これらの情報を参照してプリフェッチロードの挿入箇所を決定する。

【００１１】プリフェッチロードの数が一つであれば、これをループの先頭に配置する。複数ある場合には、展開数をその数で割って、分割された各展開の先頭にそれぞれ配置する。

【００１２】以上の処理を、既存のパイプライニング最適化処理に組み込むことで実装できるので、コンパイラの最適化処理そのものに対する負担の増加を最小限に留めることが可能である。

【００１３】

【作用】プリフェッチロードが実際にキャッシュミスを起こせば、その処理が後続の命令にオーバーラップして行われる。このため、次のキャッシュミスまでに十分な命令数があれば、発生したキャッシュミスによるプロセッサのストールを理想的にはゼロにできる。また、キャッシュミスを隠しきるだけの命令数がない場合でも、プリフェッチロード同士の距離は概ね等間隔になっているので、どのロードがキャッシュミスを起こすかが予測できず、また命令実行にオーバーラップできるキャッシュミス処理が1段しかない以上は、並列に実行できるコード部分の期待値は最大であり、何れの場合にしても、オブジェクトコードの実行性能を大幅に向上させる。

【００１４】一方、元々キャッシュに載っているデータに対するプリフェッチロードは無駄な命令になるが、プリフェッチロードの発行は、コンパイラオプションで抑止することが可能であるし、たとえ無駄になったとしても、ループ展開した上で必要最小限のプリフェッチロードしか発行していないので、性能劣化は最小に抑えることができる。

【００１５】

【実施例】以下、本発明の実施例を図に基づいて説明する。図1は本発明の一実施例を示すものである。コンパイラは図1にあるようにソースコードの字句解析・構文解析(101)、意味解析・中間コード生成(102)、大域的最適化(103)、と進行する。ここまでの処理、及び大域的レジスタ割付け(109)以降の処理は従来からのものと変わりない。また、この例では、大域的最適化の後でソフトウェアパイプライニング最適化(104)を実施している。

【００１６】本発明は、このソフトウェアパイプライニング最適化(104)の処理に改良を加える事であり、複数のプリフェッチロードを挿入する場合として、図2のコード片を例にとって説明する。このプリフェッチロード発行の最適化は、ユーザによるオプション指定があったときだけ行われ、実行されるループの回数が少ないときには適用しない。尚、この例ではfloat型は4バイト、キャッシュのライン長は32バイト、汎用レジスタ、浮動小数点数用レジスタとも32本ずつとする。ループの繰返し数Nはコンパイル時には未知であるものとする。

【００１７】ソフトウェアパイプライニング最適化処理(104)は、まず中間語の解析(105)を行い、制御変数やループの構造から、106のパイプライニング実施選択の処理で最適化を行うかどうか決定する。図2のコード片は、105で中間語をコード走査する段階までに、図3にあるような中間語になっている。走査中に制御変数を添字に持つ配列や、ループ中で更新される中身の参照を伴うポインタ変数を見つけると、これをプリフェッチロード対象としてカウントしていく。図2の例では、配列a及びcがこれに相当するので、対象は2つになる。同時にこのパスでは、変数や定数の数からレジスタの必要数も算定する。図2の例では、浮動小数点数用レジスタの方が条件が厳しく、二つのロードと掛け算のターゲット(図3、301,302)、及び変数bにレジスタが必要である。変数bはループ内では定数扱いできるので、展開数にかかわらず同じレジスタを割り付けられるため、定数としてカウントしておく。

【００１８】105で収集した情報をもとに、パイプライニング実施選択(106)の処理で、パイプライニング最適化を実施するかどうかが決定される。この結果ループが最適化の対象になった場合、これまでに集めた情報から、プリフェッチロードの挿入方法を、107で決定する。107の処理の流れを図4に示した。詳細な処理は以下[1]～[3]に示す順序で行い、107を終了した後、変換後中間コード出力(108)の処理で、プリフェッチロード指示の中間語コードを挿入したものを出力する。

【００１９】[1] 最適な展開数とレジスタ資源からくる展開数の制限値を比較して、展開方法を決定する。まず

401で最適な展開数を、キャッシュのライン長を、繰り返し一回あたりで参照されるプリフェッチロード対象領域の幅で割ることで算定する。図2の例では、ライン長が32バイトで、4バイトの配列メンバを1つずつアクセスしているので、32/4で8になる。つぎに、後方の大域的レジスタ割付け(109)で、レジスタが不足しないように、402で必要レジスタ数から展開の制限値を算定する。変数の数をv,定数の数をc,割付け可能なレジスタ数をmとすると、最大展開数は次の不等式

$$n \times v + c \leqq m$$

を満たす最大の自然数nにほかならない。図2の例では、v=3,c=1,m=32であるから最大展開数nは10となる。この結果403の比較が成立するので、最適展開数でループ展開を行う(404)。仮に不成立な場合でも、プリフェッチロードがループ展開を必要としている理由は、従来のものとは異なるので、図5にあるように可能な回数の展開を繰り返すことで、最適展開数分のコードを繰り返し1回で実行できるようする(405)。図5は、最適展開数8を4展開分のコード(501,502)を2回繰り返すことで実現している例で、501,502で同じレジスタセットが使えるようにしている。

【００２０】[2] プリフェッチロードの数から、挿入位置を決定する。まず、プリフェッチロードの数をチェック(406)して、もし1つならループの先頭に配置(407)する。プリフェッチロードが複数あるならば、展開数をプリフェッチロードの数で割って(408)、なるべく等間隔に配置する。図2の例ではプリフェッチロードが2個あるので、図6のように、4展開ずつコード配置して、各部分(602,604)の先頭に配列a,cに対応するプリフェッチロード(601,603)を挿入する。このことで、挿入されたプリフェッチロードは概ね等間隔になり、かつ、そこで命令スケジュールの単位を切るようにすれば、命令スケジューラ(図1,110)はそれ以上の意識を持たなくても、この間隔を維持することが可能である。

【００２１】尤も、この方法は実現の一例であり、命令スケジュール(110)がループ中のプリフェッチロードを等間隔に配置しなおす設計も可能である。

【００２２】[3] 次に409でプリフェッチの幅を決定する。これはハードウェア機構に大きく依存するが、通常はプリフェッチしたロケーションをアクセスするまでに＊キャッシュミス処理が終わるだけの余裕があれば十分である。ただし、もしプリフェッチロードが他のキャッシュミス処理中に現われた場合は無効命令化されるようにハードウェアサポートされていれば、1ライン長程度幅を大きめにとる。

【００２３】尚、この例では対象計算機がサポートしている命令並列性は比較的低いと仮定している。このため、ソフトウェアパイプライニングが要求するループ展開数は403の比較対象より小さいので、ここでは折り込んでいない。また、同じ理由から、大域的レジスタ割付け(109)はレジスタ数最小に割付けるものとして、必要レジスタ数を推定している。もっと高い命令並列性を持つ計算機に対しては、これらのバランスは、大きく変更されることに配慮が必要である。

【００２４】

【発明の効果】本発明によれば、プリフェッチロードの発行による最適化が、コンパイラによって自動的に実施できるため、キャッシュミスによって起こるプロセッサストールを動的に削減し、オブジェクトコードの実行性能を大幅に向上することができる。

【図面の簡単な説明】

【図１】本発明で採用したコンパイラの構成図である。

【図２】本発明の具体例を説明するためのソースプログラムを示す図である。

【図３】図２のソースプログラムに対する中間コードのモデルを示す図である

【図４】プリフェッチロードの実施を判断する部分の構成図である。

【図５】最適なプリフェッチロードの出力に必要なループ展開数がレジスタ資源の制約により確保できない場合の展開方法を示す図である。

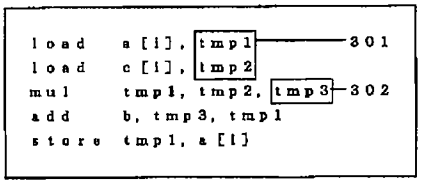
【図６】図２のソースプログラムに対する変換後中間コードのモデルを示す図である。

【符号の説明】

１０４・・・ソフトウェアパイプライニング最適化処理
１０５・・・解析
１０６・・・パイプライニング実施選択
１０７・・・プリフェッチロード実施選択
１０８・・・変換後中間コード出力

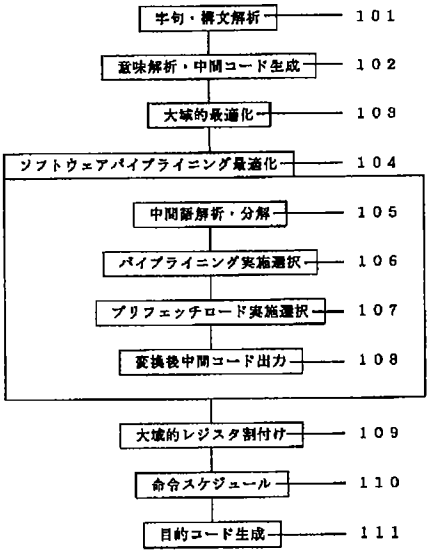
【図３】

図 3



【図1】

図 1



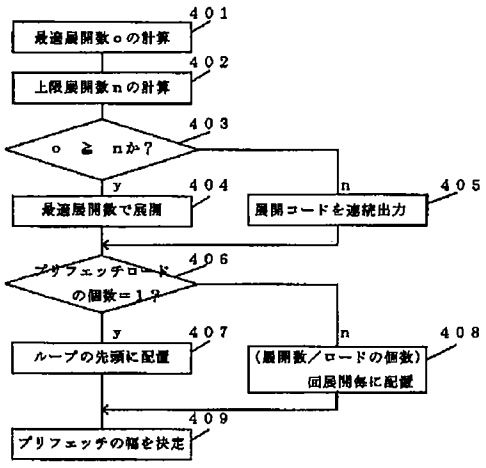
【図2】

図 2

```
float    a[N], c[N], b
int      i;
                :
                :
for  (i = 0; i < N; i++)  {
     a[i] += b * c[i];
}
                ;
```

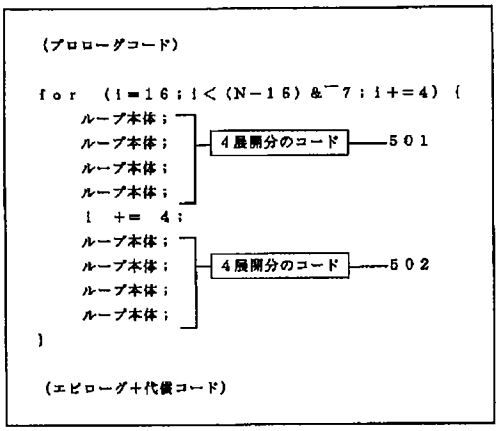
【図4】

図 4



【図5】

図 5



【図6】

図 6